EMBEDDED OPEN SOURCE SUMMIT

EMBEDDED LINUX CONFERENCE

Tweaking Device Drivers for Achieving Real-time Performance in Embedded Systems Using RT Linux

2023-06-29, Prague

Vaishnav Achath
Vignesh Raghavendra
Keerthy J

Texas Instruments

About us | TI Processors and Open source

THE LINUX FOUNDATION

Real-Time LINUX

yocto PROJECT

Decades of contribution and collaboration

Ingrained culture to give back to the community

Upstream FIRST!

Focus on long term, sustainable and quality products

Upstream and opensource ecosystem in device architecture

Upstream FIRST mentality!

Speakers | Intro

Vaishnav Achath, Software Engineer at Texas Instruments India. Vaishnav works on Linux Kernel and U-Boot as part of the Texas Instruments Linux development team. Vaishnav is also a maintainer for TI platforms in Zephyr RTOS.

Vignesh Raghavendra, Software Engineer at Texas Instruments India. Vignesh co-maintains the TI's arm64 SoCs in mainline along with a few drivers. He has been contributing to Linux Kernel and U-Boot since 2014 as part of Texas Instruments' Linux development team

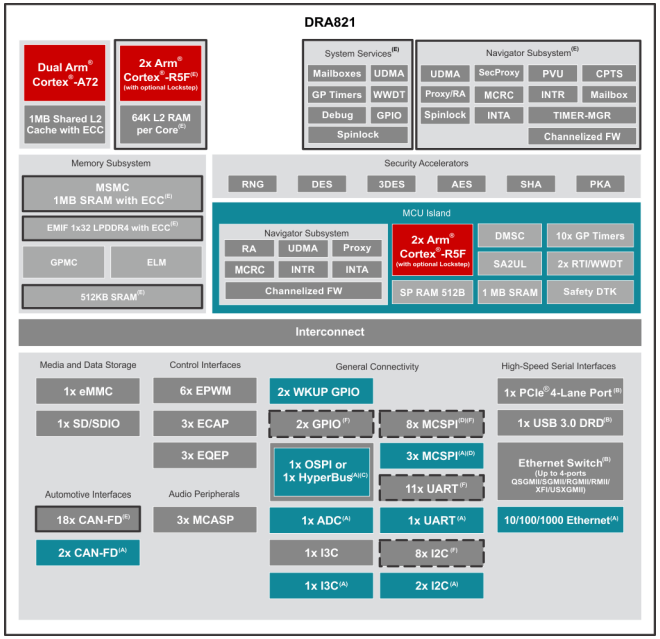
Keerthy J, is a SW Application Engineer with Texas Instruments Inc., as part of this role he primarily interacts with customers regarding their use cases in automotive and industrial applications.

Overview

- Problem Statement
- RT Linux – expectations.
- Generic tweaking options in RT Linux.
- SPI Client support and Challenges.
- Subsystem-level tweaking options for the predictability.
- Profiling and improving device drivers.
  - Case study: SPI + DMA
- Results and best practices.

Problem Statement | SPI

- **DRA821** is a heterogenous multicore application processor from Texas Instruments with Dual Arm Cortex-A72 and quad Cortex-R5F.
- DRA821 consists of 11 instances of MCSPI (Multi-channel Serial Peripheral Interface), MCSPI can operate in:
  - Full duplex Host mode, Target mode.
  - DMA mode (system DMA).

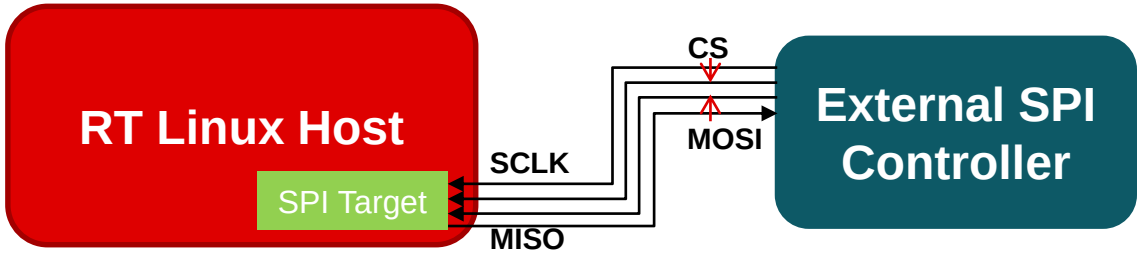


Problem:

- **MCSPI in Target mode on RT-Linux in DMA mode provides unreliable performance and observed packet loss issues (800 ms/128-Byte full duplex).**
- **MCSPI in Host mode on RT-Linux in DMA mode results in non-deterministic performance/ latency spikes (5 ms).**

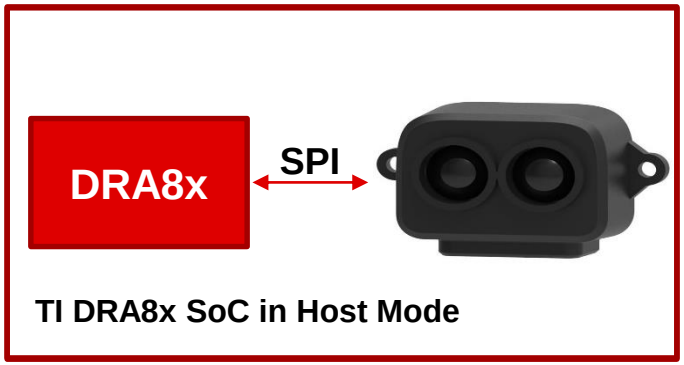
RT Linux | User Expectations

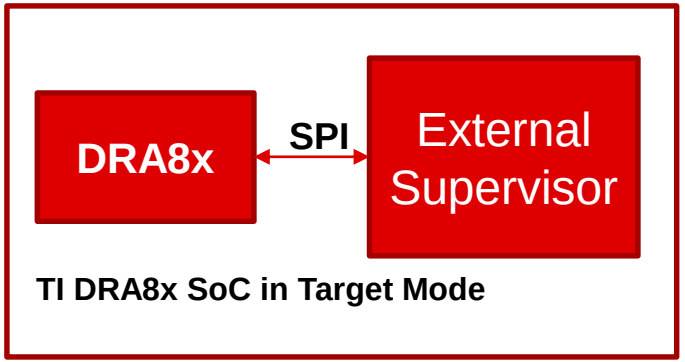
- RT Linux is increasingly becoming popular for real-time embedded applications due to the:
  - Versatility and flexibility provided by the High-Level OS.
  - Real-time capabilities and deterministic performance.
- Demand deterministic performance, when interfacing with external peripherals.
- Predictable performance is critical in cases where the RT Linux host is acting in target mode and an external device initiates and controls the communication.
- CAN, SPI, and UART are popular media communication media with RT requirements



Practical usage | Host and Target

- Industrial robotic application | SPI Host
  - LIDAR sensors that can be interfaced to host through SPI, in such use-cases deterministic capture of sensor data for critical for the system
  - Robotic motor control (non-safety critical)
- Device management interface | SPI target
  - External supervisor interaction.
  - Control, monitor, and power state.
  - Firmware management.
  - Watchdog-like ping-pong.



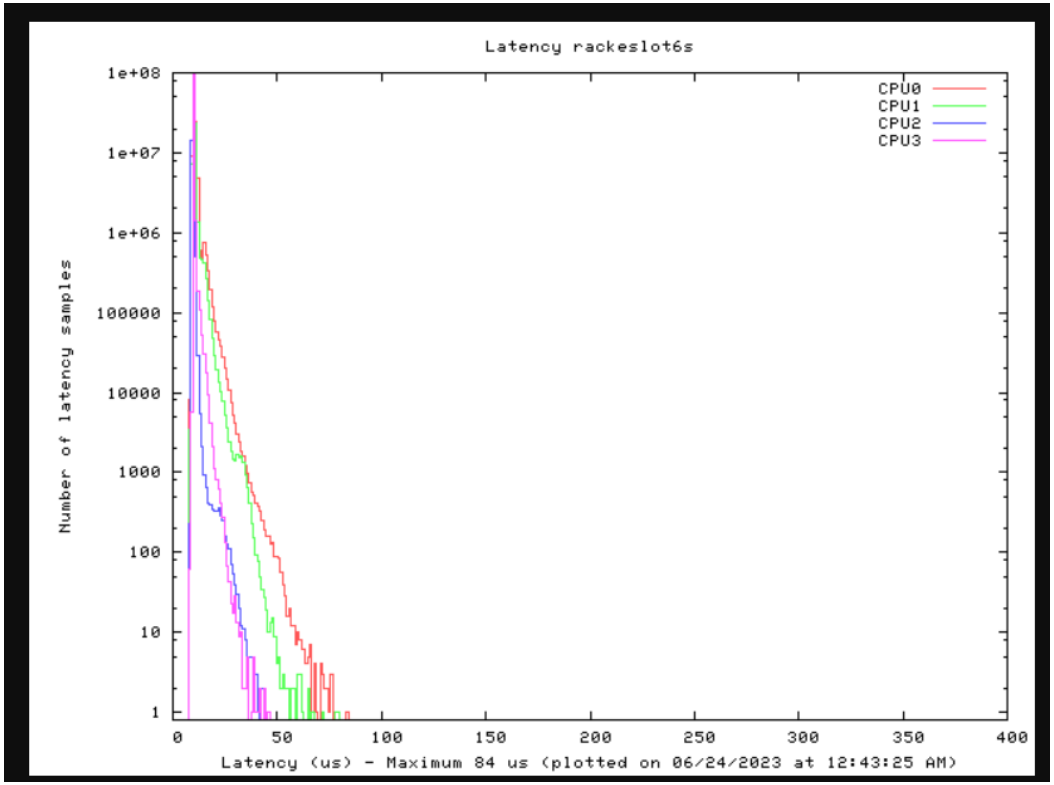


RT Linux tuning | Where to start

- cyclictest
  - measure a thread's intended wake-up time and the time at which it actually wakes up
  - Provides overheads to take into account for full RT stack
  - More at https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

```
cyclictest --mlockall --smp --priority=99 --interval=200 –loops=100000000
```

RT Linux tuning | cyclictest



HW:
TI AM625 SoC

https://www.osadl.org/Latency-plot-of-system-in-rack-e-slot.qa-latencyplot-res6.0.html?shadow=1

RT Linux | Tools

- `lmbench` – DDR bandwidth and latency analysis
  - Provides bounds on HW capability and any bottlenecks in HW leading latency excursions.
- Tracing tools: `ftrace, perf`
  https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/start
- rtla-timerlat
  - https://docs.kernel.org/tools/rtla/rtla-timerlat.html

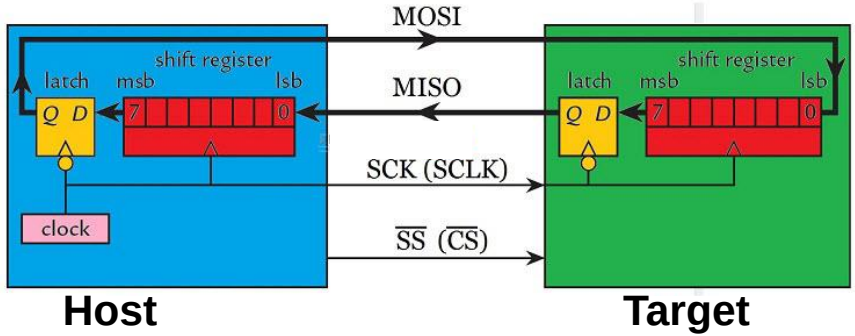
RT Linux | System Tuning

- RT Linux offers multiple tuning knobs:
  - Kernel config options (PM, CPUFREQ, CONFIG_DEBUG*, and any unneeded configs)
  - Real-time policies (SCHED_FIFO etc)
  - Real-time kernel thread priorities
  - Task partitioning (isolcpus, taskset, IRQ affinity, load balancing)
  - Modifying real-time priority of application programs .etc. (nice, chrt)
  - Latency optimizations: https://wiki.linuxfoundation.org/realtime/documentation/howto/debugging/start

- Even with all the tweaking options available:
  - device drivers need optimization for deterministic performance, to benefit from the RT Linux kernel.
  - Debugging real-time issues with device drivers can be complex especially when multiple subsystems interact. (E.g. SPI + DMA + user application)

SPI Host / Target | Linux Support

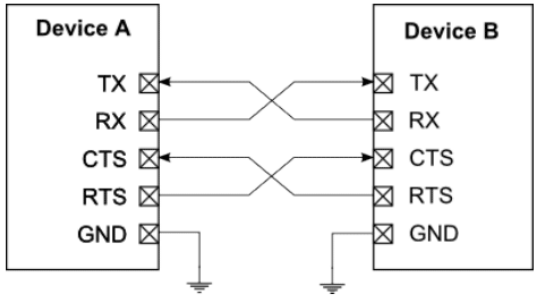
- SPI is a full-duplex synchronous serial communications module. SPI provides a cost-effective way to interface serially with external peripherals.
- SPI operates in either **Host** mode or **Target** mode.
  - In host mode, the SPI controller generates the synchronous communication clock(SCLK) and initiates the transaction.
  - In target mode, the controller receives the SCLK as input and completes the transaction initiated by the external host, the client has no control over the transaction start, speed, etc.



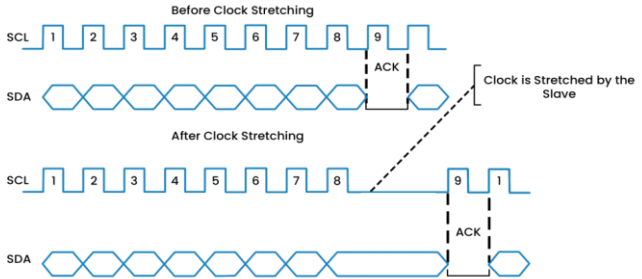
- SPI Target mode support is available in the Linux kernel SPI subsystem from v4.13 onwards and 6 SPI controller drivers support Target mode as of v6.4.

SPI Target mode | Challenges

- Full duplex: simultaneous transmit and receive.
- Host has control: target implementation demands hard real-time.
- Target must have prepared the TX transaction before the Host starts the transfer.
- Target response cannot depend on the Host request in the same message.
- Lack of **standard HW flow control** mechanism.



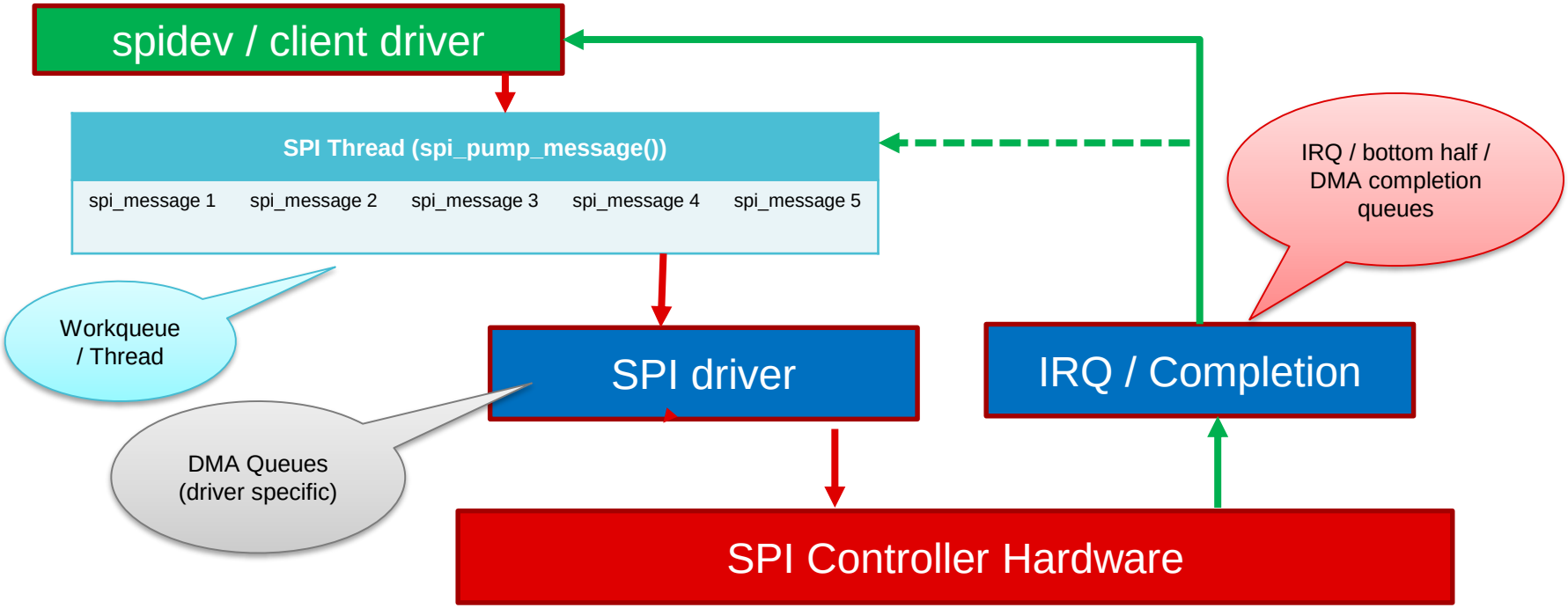
UART Flow control



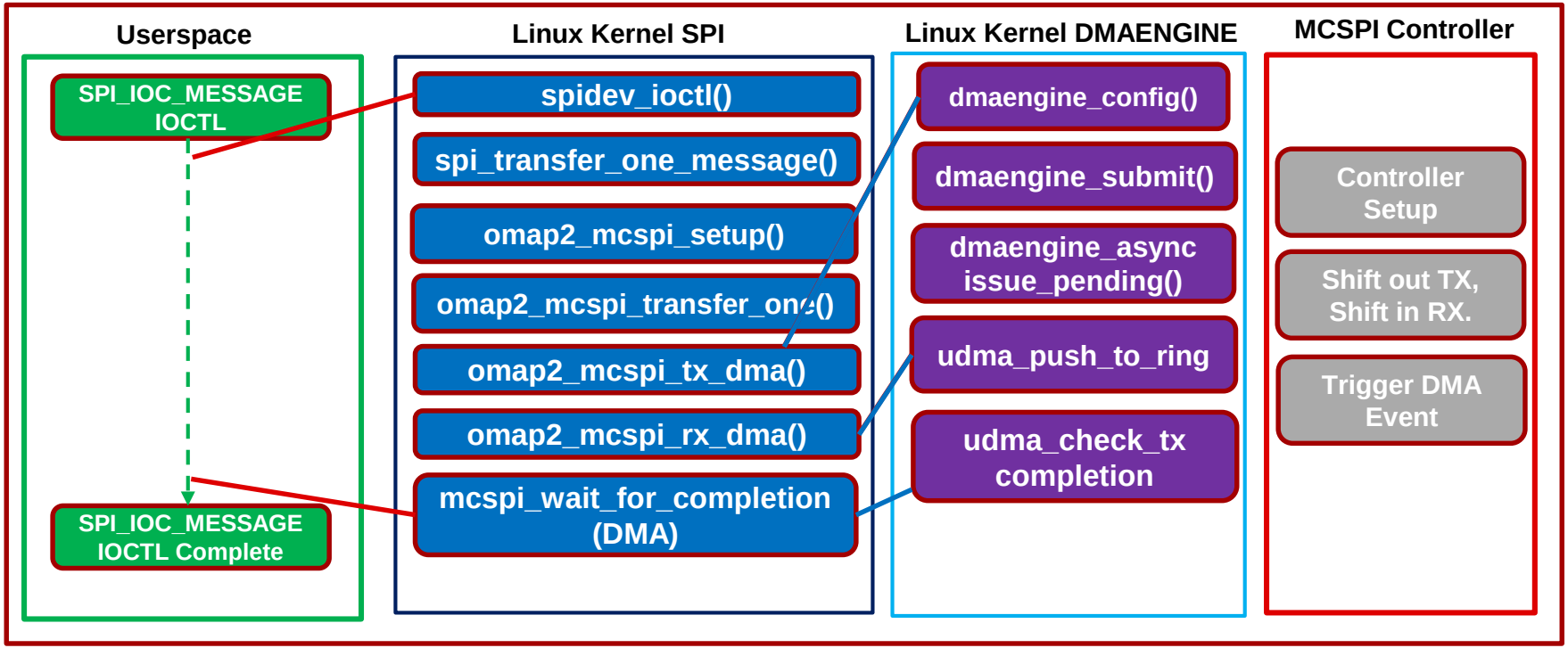
I2C Clock stretching

Courtesy: Linux as an SPI Target, Geert Uytterhoeven

Analyzing subsystem | SPI framework



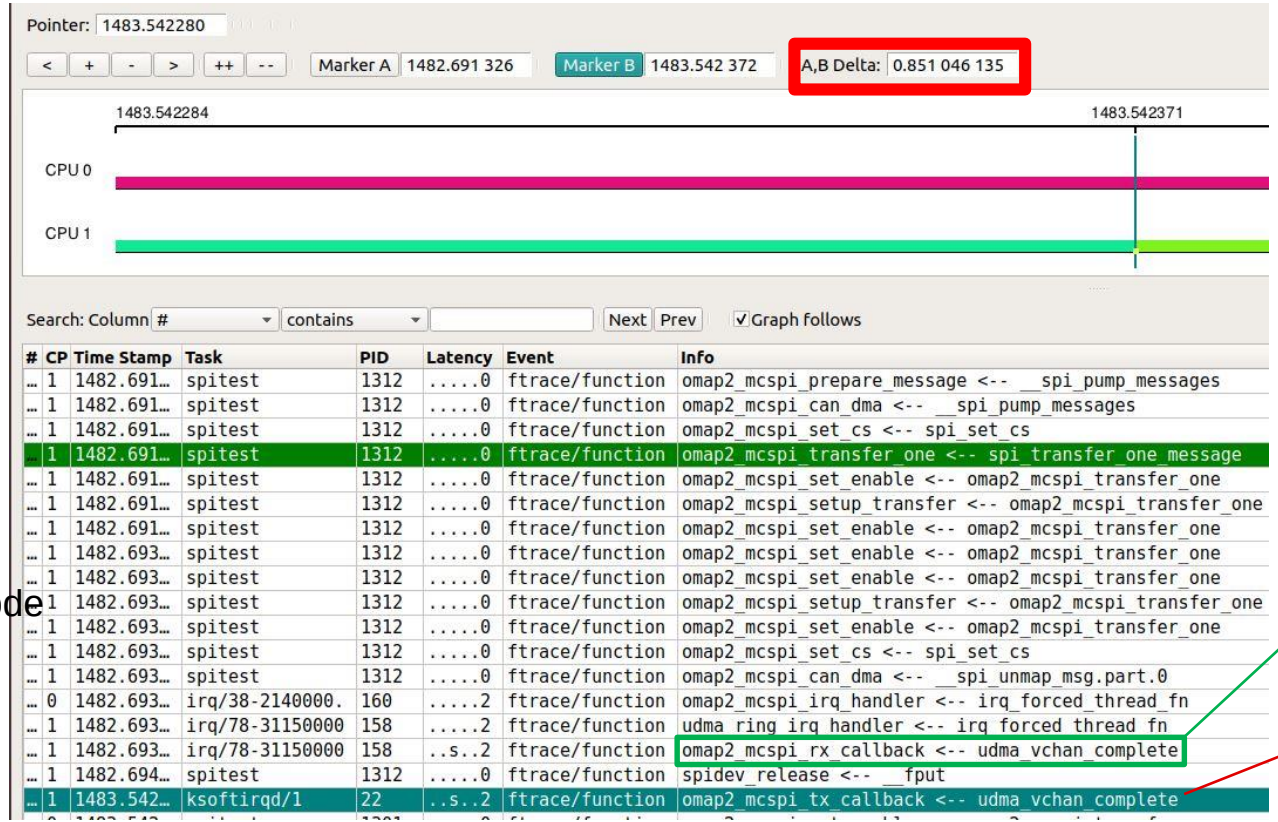
Full call stack | SPI and DMA interaction



Call Stack for SPI with DMA on DRA8x

Kernelshark Traces | Actual trace screenshots

SPI in Target mode with DMA

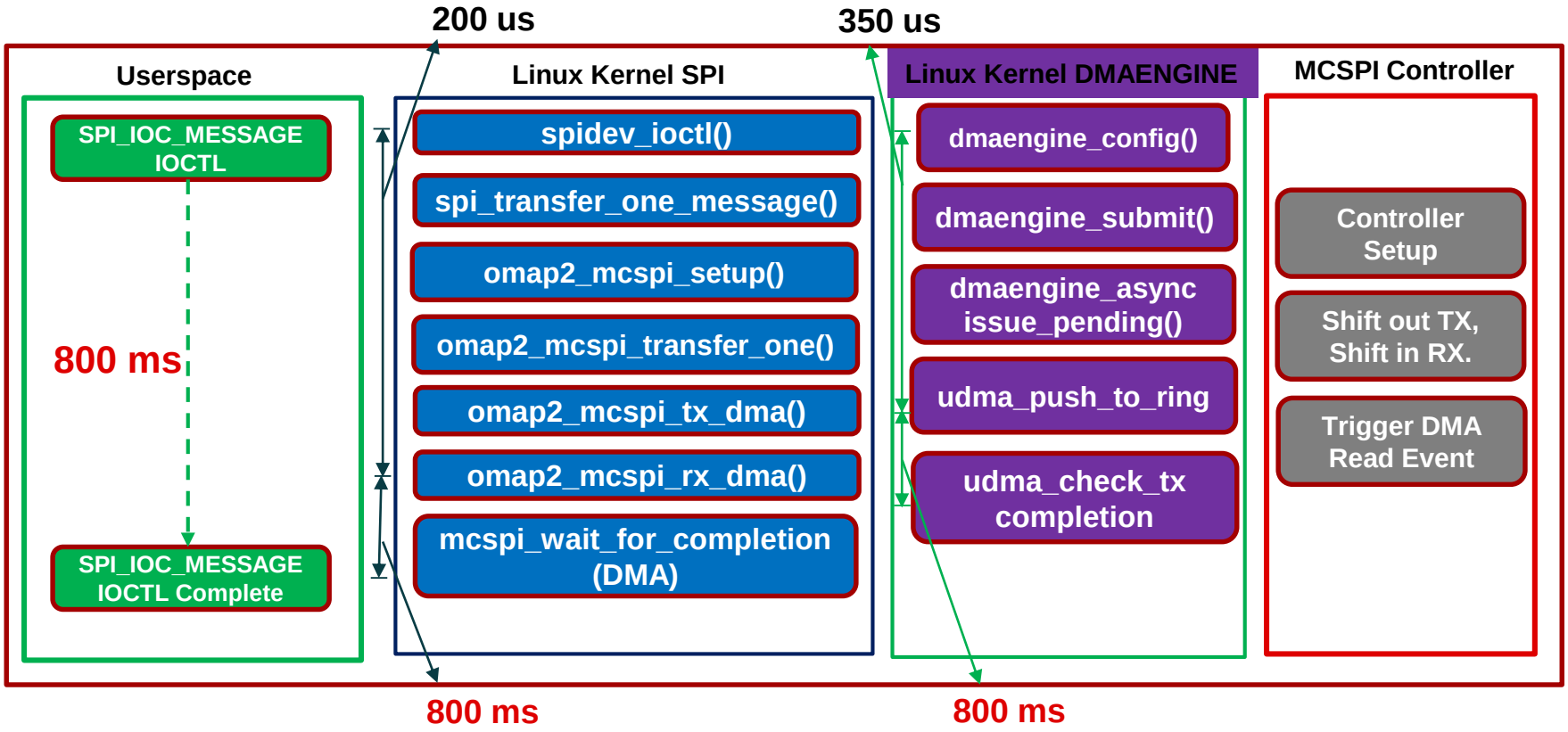


#	CP	Time Stamp	Task	PID	Latency	Event	Info
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_prepare_message <-- __spi_pump_messages
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_can_dma <-- __spi_pump_messages
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_set_cs <-- spi_set_cs
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_transfer_one <-- spi_transfer_one_message
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_setup_transfer <-- omap2_mcspi_transfer_one
…	1	1482.691…	spitest	1312	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	1482.693…	spitest	1312	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	1482.693…	spitest	1312	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	1482.693…	spitest	1312	0	ftrace/function	omap2_mcspi_setup_transfer <-- omap2_mcspi_transfer_one
…	1	1482.693…	spitest	1312	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	1482.693…	spitest	1312	0	ftrace/function	omap2_mcspi_set_cs <-- spi_set_cs
…	1	1482.693…	spitest	1312	0	ftrace/function	omap2_mcspi_can_dma <-- __spi_unmap_msg.part.0
…	0	1482.693…	irq/38-2140000.	160	2	ftrace/function	omap2_mcspi_irq_handler <-- irq_forced_thread_fn
…	1	1482.693…	irq/78-31150000	158	2	ftrace/function	udma_ring_irq_handler <-- irq_forced_thread_fn
…	1	1482.693…	irq/78-31150000	158	..s..2	ftrace/function	omap2_mcspi_rx_callback <-- udma_vchan_complete
…	1	1482.694…	spitest	1312	0	ftrace/function	spidev_release <-- __fput
…	1	1483.542…	ksoftirqd/1	22	..s..2	ftrace/function	omap2_mcspi_tx_callback <-- udma_vchan_complete

RX callback within 2ms

TX callback takes 850ms

Stack Latency | Trace summary



Tweaking options | SPI subsytem

- SPI Controller configuration can indicate if this controller should run the message pump thread with high (real-time) priority.
- SPI devices can also request real-time priority during transaction setup which will modify the controller message pump thread to real-time priority.
- This tweaking option might also not help in cases where the controller driver is not optimized.

include/linux/spi/spi.h:

```
struct spi_controller {
    …
    bool rt;
};


struct spi_device {
    …
    bool rt;
};
```

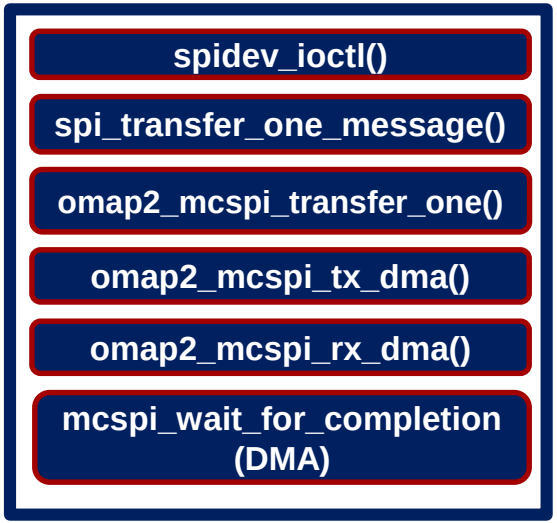
drivers/spi/spi.c:

```
if (controller->rt)
  spi_set_thread_rt(struct spi_controller *ctlr):

static void spi_set_thread_rt(struct spi_controller *ctlr)
{
        ...
        sched_set_fifo(ctlr->kworker->task);
}
```

SPI Controller driver | Analysis

- In the MCSPI controller driver, with DMA enabled, we perform the SPI controller setup, queue the TX and RX DMA operations once the message is transferred, and wait for DMA completion.



Observations

- First transaction initiated from userspace never has data mismatch/latency issues in client mode.
- When multiple transactions are queued from userspace if the delay between each transaction is small(<100 ms), client packets RX and TX are corrupted in client mode.
- In target mode

DMA driver analysis | Finding Bottlenecks

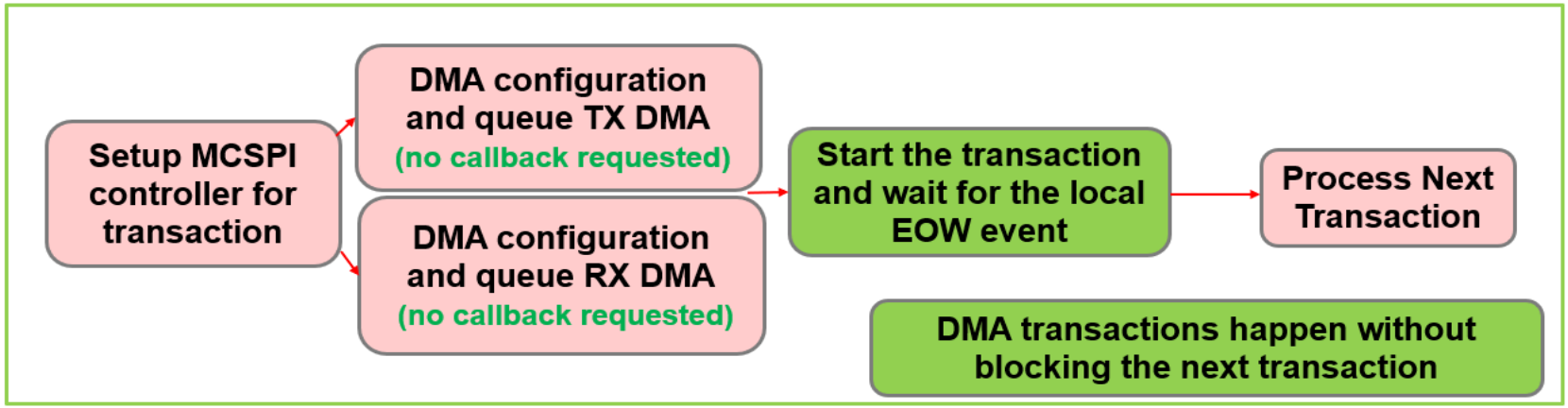
- Typical Driver design to Queue DMA transfers
  - Submit xfer(): `dmaengine_submit(), dmaengine_async_issue_pending()`
  - `wait_for_completion()` : Wait for `transfer->callback()` -> dmaengine completion callback
- Tasklet calls driver's completion callback
  - Per channel tasklet scheduled on completion
  - Multiple tasklets (depending on channel usage) can end up with similar priorities
  - Callback function needs to be light-weight for better response
  - Sched priority of each tasklets needs to be tuned
- DMA Subsystem on DRA8x/K3 SoCs have a Network on Chip arch
  - Centralized DMA talks to mini DMAs closer to peripherals (k3-udma.c)
  - Deferred workqueue handler (`udma_check_tx_completion()`) checks the remote DMA states to ensure the pipe is flushed before calling driver callbacks
- Above tasklet and deferred workqueue causes all the jitter in the SPI transfer RT path

Bottleneck | Analysis and Fixes

- Latency graphs indicated 3 issues
  - spi_pump_message
  - DMA tasklet for signaling completion
  - DMA completion / threaded IRQ handler inside DMA driver (udma_check_tx_completion())
- **Solution**
  - Setup spi_pump_message as RT (`spi_controller->rt = true`)
  - Set DMA tasklet as realtime and higher priority relative to other task pool
  - Convert workqeueue within DMA driver to be RT priority
- This reduces the jitter and brings in determinism
  - Did not meet overall latency goal as even with tasklets, there was enough contention for large excursions.

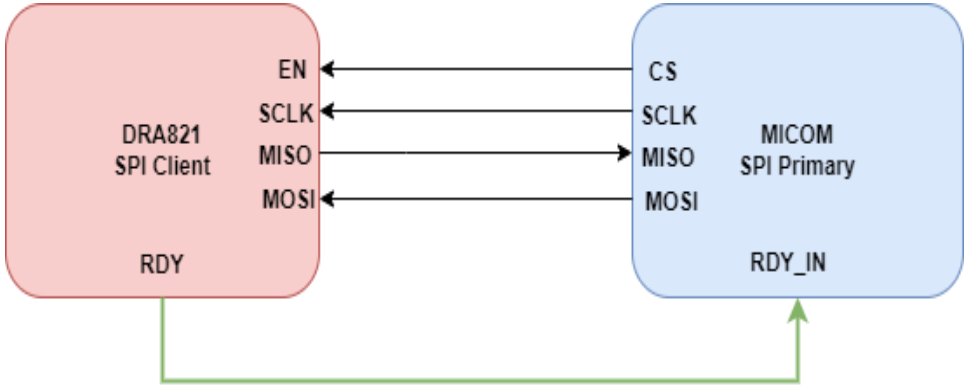
Driver updates | Eliminate DMA tasklets from RT path

- McSPI controller provides a word count interrupt (End of Word), end of transfer
- Use McSPI to packetize transfers: provides interrupt after programmed number of words are clocked in/out.
- Use the same for continuing to the next transaction to avoid delays in the tasklets.
- Can be used if host and target use fixed-size packets and negotiated in advance.
- Multiple packets are needed in case of variable size.

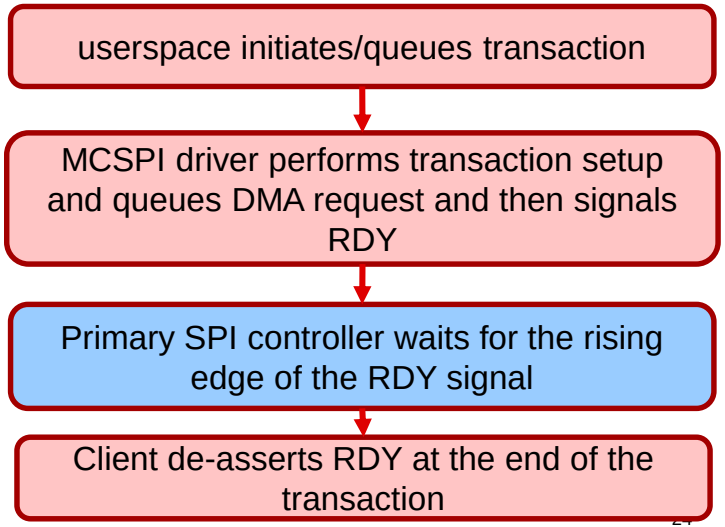


Enhancements | GPIO-based flow control

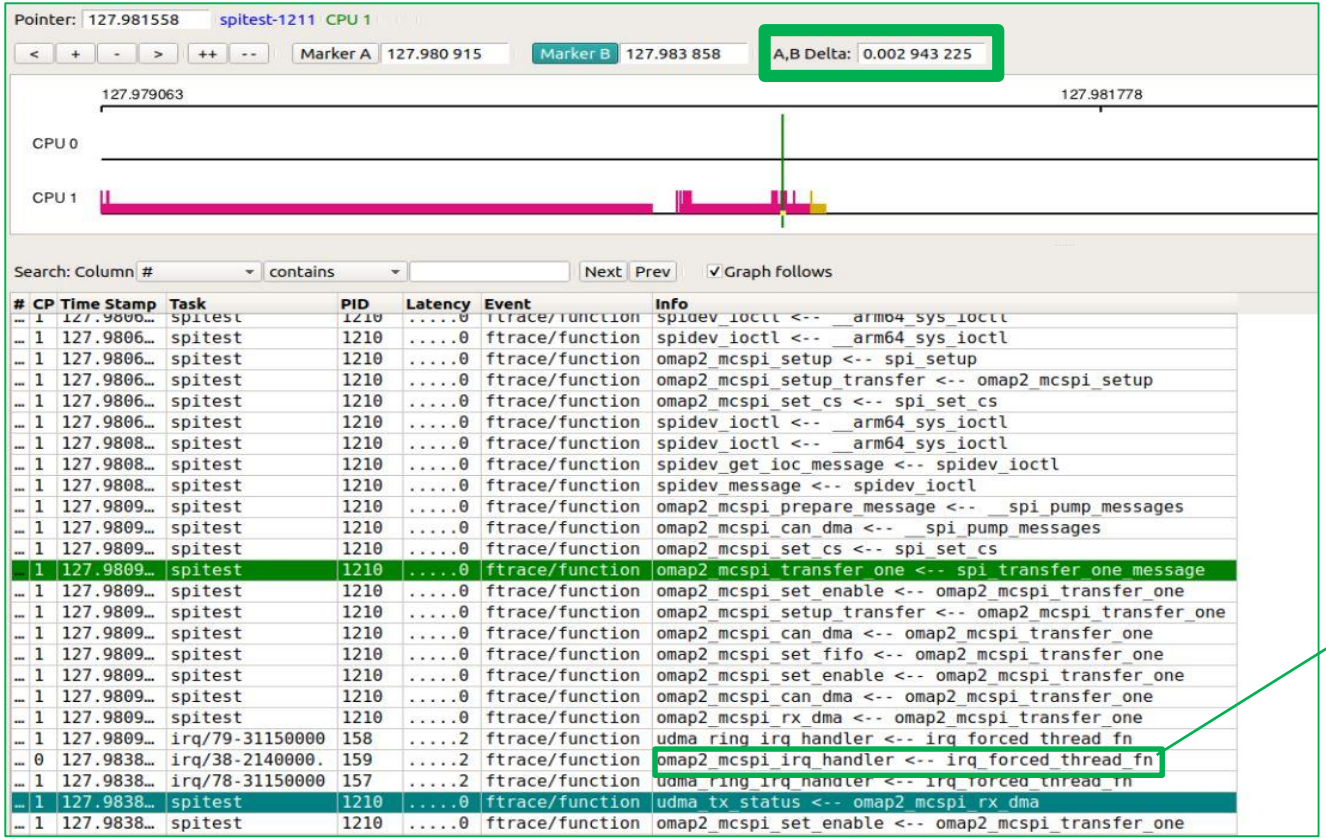
- Bounded latency possible with RT Linux
  - During high IRQ and RT load conditions, the performance of the SPI client deteriorates
  - For non-RT systems, implement GPIO-based flow control
  - Single RDY GPIO was implemented to achieve reliability requirements



RDY GPIO based Flow Control Mechanism



After optimization | kernel shark trace

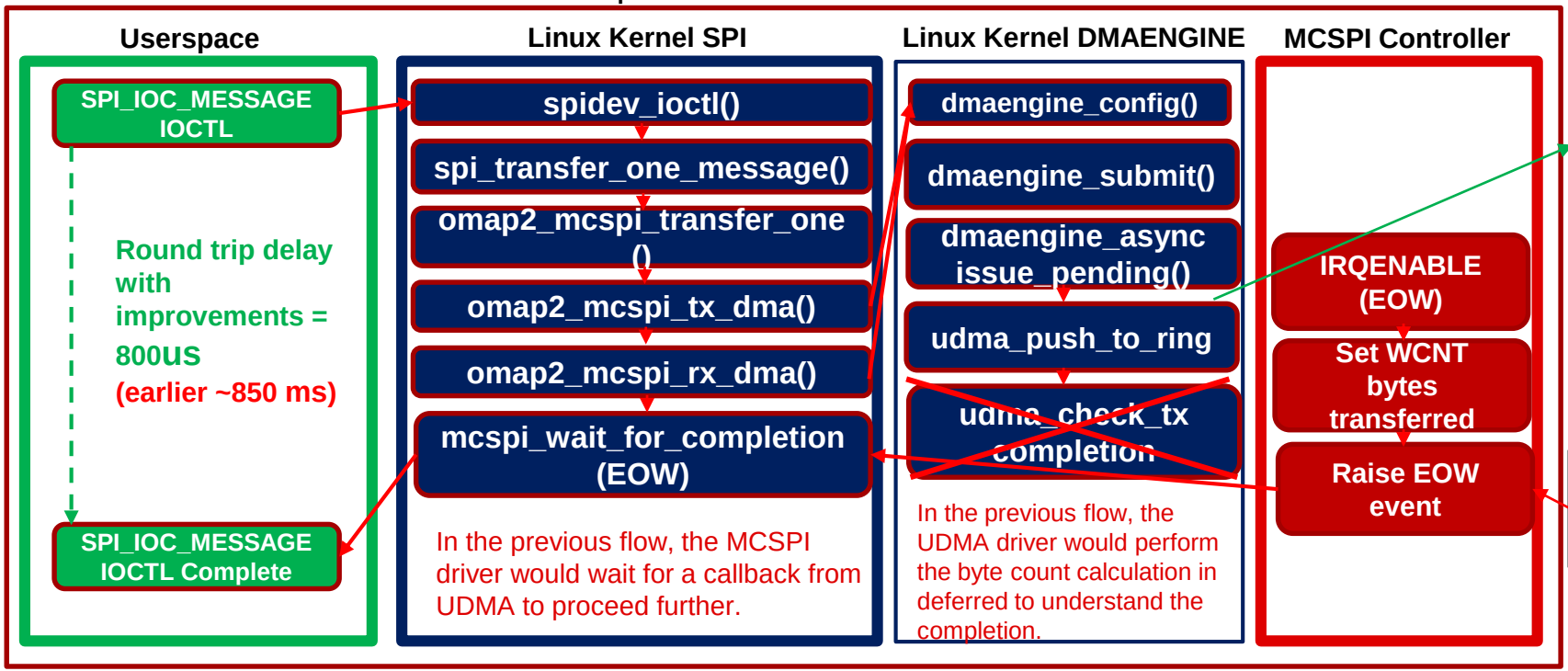


#	CP	Time Stamp	Task	PID	Latency	Event	Info
…	1	127.9806…	spitest	1210	0	ftrace/function	spidev_ioctl <-- __arm64_sys_ioctl
…	1	127.9806…	spitest	1210	0	ftrace/function	omap2_mcspi_setup <-- spi_setup
…	1	127.9806…	spitest	1210	0	ftrace/function	omap2_mcspi_setup_transfer <-- omap2_mcspi_setup
…	1	127.9806…	spitest	1210	0	ftrace/function	omap2_mcspi_set_cs <-- spi_set_cs
…	1	127.9806…	spitest	1210	0	ftrace/function	spidev_ioctl <-- __arm64_sys_ioctl
…	1	127.9808…	spitest	1210	0	ftrace/function	spidev_ioctl <-- __arm64_sys_ioctl
…	1	127.9808…	spitest	1210	0	ftrace/function	spidev_get_ioc_message <-- spidev_ioctl
…	1	127.9808…	spitest	1210	0	ftrace/function	spidev_message <-- spidev_ioctl
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_prepare_message <-- __spi_pump_messages
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_can_dma <-- __spi_pump_messages
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_set_cs <-- spi_set_cs
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_transfer_one <-- spi_transfer_one_message
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_setup_transfer <-- omap2_mcspi_transfer_one
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_can_dma <-- omap2_mcspi_transfer_one
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_set_fifo <-- omap2_mcspi_transfer_one
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_can_dma <-- omap2_mcspi_transfer_one
…	1	127.9809…	spitest	1210	0	ftrace/function	omap2_mcspi_rx_dma <-- omap2_mcspi_transfer_one
…	1	127.9809…	irq/79-31150000	158	2	ftrace/function	udma_ring_irq_handler <-- irq_forced_thread_fn
…	0	127.9838…	irq/38-2140000.	159	2	ftrace/function	omap2_mcspi_irq_handler <-- irq_forced_thread_fn
…	1	127.9838…	irq/78-31150000	157	2	ftrace/function	udma_ring_irq_handler <-- irq_forced_thread_fn
…	1	127.9838…	spitest	1210	0	ftrace/function	udma_tx_status <-- omap2_mcspi_rx_dma
…	1	127.9838…	spitest	1210	0	ftrace/function	omap2_mcspi_set_enable <-- omap2_mcspi_transfer_one

completion within 2.9 ms

Solution

The following flow diagram shows the complete flow of operations and optimizations made with GPIO-based flow control for a full duplex transfer:



Results before and after

Target Mode

Transaction Type	After Optimization	Before Optimization
128 byte (full duplex)	800 us per transaction	800 ms per transaction
160 byte (full duplex)	800 us per transaction	800 ms per transaction
128 byte (full duplex) with flow control under 99% CPU load stress	2 ms per transaction	800 ms per transaction
640 Byte (full duplex) with flow control	2 ms per transaction	800 ms per transaction

Host Mode

Transaction Type	After Optimization	Before Optimization
160 byte (full duplex)	Peak latency 400 us	Peak latency as high as 5 ms.

Best practices | Driver design

- Tune the system level latency first
  - `cyclictest, rtla-timerlat, lmbench` with the minimal system (no drivers, features, etc).
- Keep the IRQ-off state to a minimum.
- Beware of priority inversion-like situations.
  - RT task depending on low priority workqueues, threads to proceed further.
  - Multi-level workqueues may kill the determinism.
- Complex SW state machines make it harder to avoid latency
  - Rely on HW state as much as possible.
  - Feed back to HW designers if such support is missing.

References

- Linux as an SPI Target, Geert Uytterhoeven, FOSSDEM 2018.
- Adventures In Real-Time Performance Tuning, Frank Rowand.
- https://www.linkedin.com/pulse/checklist-real-time-applications-linux-linutronix/
  - Linuxtronix,
- DRA821 Product details, https://www.ti.com/product/DRA821U

- Details and Instructions:
  - https://github.com/vaishnavachath/elc-eoss23-rtlinux

Credits and Acknowledgement

- Texas Instruments Inc.
- The Linux Foundation.

Q&A

- Contact Information:
  - Vaishnav Achath <vaishnav.a@ti.com>
  - Vignesh Raghavendra <vigneshr@ti.com>
  - Keerthy J <keerthy@ti.com>
- Also on IRC @ libera.chat #linux-ti

Learn more about TI products

- https://www.ti.com/linux
- http://opensource.ti.com/
- https://www.ti.com/processors
- https://www.ti.com/edgeai